For TRS-80

MICROSOFT

muMATH

muMATH

MICROSOFT

MICROSOFT

muMATH 1
FOR TRS-80

MATH32

Catalog No. 1208 Part No. 13H08

MICROSOFT
CONSUMER PRODUCTS
400 108th Ave. N.E., Suite 200
Bellevue, WA 98004

MICROSOFT

muMATH

muMATH / muSIMP

Produced by Microsoft
Written by The Soft Warehouse
Instruction Booklet by William Barden Jr. and Gregory Whitten

Microsoft Consumer Products
400 108th Ave. NE, Suite 200
Bellevue, WA 98004

COPYRIGHT NOTICE

Microsoft muMATH/muSIMP is licensed from The Soft Warehouse, Honolulu, Hawaii and is copyrighted under United States Copyright laws by Microsoft.

It is against the law to copy muMATH/muSIMP on cassette tape, disk, or any other medium for any purpose other than personal convenience.

It is against the law to give away or resell copies of Microsoft muMATH/muSIMP. Any unauthorized distribution of this product deprives the authors of their deserved royalties. Microsoft will take full legal action against violators.

If you have questions on this copyright, please contact:

Microsoft Consumer Products
400 108th Ave. NE, Suite 200
Bellevue, WA 98004

Table of Contents

Chapter One
Some Introductory Notes on Using muMATH/muSIMP

Chapter Two
Arithmetic Operations Using muMATH

Chapter Three
Using muMATH to Solve Algebraic Problems

Chapter Four
muMATH in Higher-Level Math Processing

Chapter Five
Programming in muSIMP and TRS-80 Functions

Chapter One
Some Introductory Notes on Using muMATH/muSIMP

- What is muMATH/muSIMP?
- The Designers
- A Word About Microsoft
- The Right Hardware
- The muMATH/muSIMP Diskette
- Making a Backup Copy of muMATH/muSIMP
- How To Load muMATH/muSIMP
- muMATH/muSIMP Line Input and Display

What is muMATH/muSIMP?

muMATH/muSIMP is a software package, supplied on diskette, that implements an exciting new symbolic math system on the TRS-80. The TRS-80 user can now enter equations directly into muMATH in a "calculator mode" to solve problems in arithmetic, algebra, trigonometry, and calculus without first having to "program" the problem in a computer language. In addition to allowing direct entry of equations, muMATH provides such powerful features as exact representation of results to any number of digits, automatic reordering and simplification of terms, and collection of similar terms.

muMATH is ideally suited for interactive, practical use by engineers, scientists, and mathematicians. It is also an excellent **self-teaching tool** for students interested in the standard mathematics curriculum from elementary arithmetic through calculus.

The muSIMP portion of the package may be used to program new functions to extend the capabilities of the muMATH/muSIMP system. However, the programming mode is not a requirement for using this package.

muMATH/muSIMP is designed to be used on a TRS-80 Model I computer with either 32K or 48K of RAM memory and one disk drive.

As a brief sample of the capabilities of muMATH, consider the following problems:

Problem: Find the volume in cubic inches of an auditorium 211 feet long by 153 feet wide by 76 feet high.

Answer: Entering

```
211*12*153*12*76*12;
```

results in an immediate printout of

```
@ 4239661824
```

Problem: Find the 1023rd power of 2.

Answer: Entering

```
2↑1023;
```

results in the exact display after about 10 seconds of

```
@ 898846567431157953864652595394512366808988489471153
2863671504057886633790275048156635423866120376801056 0
0569399356966788293948844072083112464237153197370621 8
8883946712432742638151109800623047059726541476042502 8
8441907534117123144073695655527041361858167525534229 3
1491199736229692398581524176781648121120686 08
```

Problem: Find the binomial expansion of (A+B) ↑5.

Answer: Entering

```
PWREXPD:2; (A+B)↑5;
```

results in a display after about eleven seconds of

```
@ 5*A*B↑4 + 10*A↑2*B↑3 + 10*A↑3*B↑2 + 5*A↑4*B +A↑5
+ B↑5
```

Problem: Integrate the expression (2X–1/X) with respect to X.

Answer: Entering

```
INT (2*X—1/X, X);
```

results in a display after about four seconds of

```
@ X↑2 – LN(X)
```

We'll be presenting more of the versatile capabilities of muMATH and muSIMP in the following chapters. The people at Microsoft hope you'll be pleased with the muMATH/muSIMP package, and know you will find it a powerful tool in dealing with mathematical applications.

The Designers

The muMATH/muSIMP package was designed and implemented by David Stoutemyer and Albert Rich of The Soft Warehouse, Honolulu, Hawaii.

The project was begun in 1976 by Albert Rich with the design and coding of a LISP language interpreter for use on 8080 and Z-80 based microprocessors. In 1977, in collaboration with David Stoutemyer, the

interpreter was upgraded in both speed and versatility. The symbolic processing capability was enhanced with the addition of infinite precision arithmetic.

Finally, in 1978, using the LISP system as a basis, muSIMP-79 was created to give the user the power of an applicative language but with a more natural syntax than LISP.

Convinced of the power and utility of symbolic mathematics, Stoutemyer and Rich developed the muMATH-79 system for microcomputers. The TRS-80 version is the culmination of this idea. Gregory Whitten of Microsoft was instrumental in implementing muMATH and muSIMP on the TRS-80 computer.

A Word About Microsoft

Microsoft produces high-quality, concise software for today's microprocessors.

Microsoft's BASIC interpreter, in its several versions, has become the standard high-level programming language used in microcomputers. In addition to Radio Shack TRS-80 Level II BASIC, and TRS-80 Disk BASIC, Microsoft has supplied BASIC Interpreters for the Commodore PET, the Apple II Computer, NCR 7200, Compucolor II, OSI, Pertec Altair, and many others.

Microsoft's careful approach to the development of microprocessor software has allowed the production of large amounts of bug-free, well-designed code in a minimum amount of time. Currently available: BASIC interpreters for the 8080, 6800, and 6502 microprocessors; a FORTRAN compiler, assembler, loader, and runtime library package for the 8080 and Z-80 microprocessors; an ANSI-74 COBOL compiler and a BASIC compiler for the 8080 and Z-80; and a complete offering of systems software for the new 16-bit microprocessors.

Microsoft Consumer Products was founded as a division of Microsoft in the summer of 1979 to provide microcomputer users with high-quality system and utility software as well as application software.

muMATH/muSIMP is just one of many Microsoft products being planned for the end-user consumer market. All of these software packages will be marketed by Microsoft Consumer Products.

Microsoft Consumer Products is dedicated to providing only the best, most reliable microcomputer software.

For more information on Microsoft Consumer Products software, please write to:

Microsoft Consumer Products
400 108th Ave. NE, Suite 200
Bellevue, WA 98004

The Right Hardware

muMATH/muSIMP can be used with the Radio Shack TRS-80 Model I Microcomputer with 32K or 48K of RAM memory and one or more disk drives.

The muMATH/muSIMP Diskette

The muMATH/muSIMP diskette that comes in your muMATH/muSIMP package is a high-quality copy of the muMATH/muSIMP system from Microsoft. **Please observe the following precautions when handling the diskette:**

1. Always place the diskette back in the jacket after use.
2. Never touch the diskette surface through the diskette window.
3. Keep the diskette away from sources of magnetism or heat such as direct sunlight.
4. Use felt-tip pens rather than hard-point pens when writing on diskette labels.
5. **NEVER** turn the TRS-80 system on or off unless all diskettes are removed from all drives.

Diskette Replacement. Your muMATH/muSIMP is guaranteed to be a faultless recording. If the diskette fails to work properly when first opened, return it to the dealer from whom it was purchased or mail the diskette with sales receipt and explanatory letter to Microsoft Consumer Products. It will be replaced at no charge.

If for any reason your diskette becomes damaged, we will replace it for a nominal $7.50 charge. Mail the diskette with your payment to Microsoft Consumer Products.

Returns should be sent to:

Microsoft Consumer Products
400 108th Ave. NE, Suite 200
Bellevue, WA 98004

Diskette Contents. All files on the diskette are standard Radio Shack TRS-80 Disk Operating System (TRSDOS) files.

Two of the files are versions of muMATH that are designed for either a 32K RAM system (MATH32/CMD) or 48K RAM system (MATH48/CMD). These are "precompiled" versions of muMATH that may be simply loaded and executed by typing "MATH32" or "MATH48" while in TRSDOS.

A third file is a demonstration file meant to show the capabilities of muMATH (DEMO/INT). This file is loaded while executing muMATH (MATH32 or MATH48).

Additional files on the diskette are "source" files for muMATH that extend the capabilities of the basic muMATH system. These files may be loaded by a special muMATH command to implement muMATH features, and their use is discussed in the following chapters.

Making a Backup Copy of muMATH/muSIMP

The first action that should be taken upon receiving muMATH/muSIMP is to produce a "backup" copy of the muMATH/muSIMP diskette. The original diskette may then be set aside and the copy used in its place. Remember, if a backup copy is not used, it is possible to destroy the muMATH/muSIMP files on the original through hardware malfunction or "operator error." Always have a backup!

Perform the following steps to produce a backup copy of muMATH/muSIMP:

1. Turn on the TRS-80 system with no diskettes in any of the drives.

2. Place a protective tab over the "write-protect" notch of the muMATH/muSIMP diskette, if one is not already there.

3. Place the muMATH/muSIMP diskette into drive 0 and "boot up" the system by a RESET or power up.

4. Set aside a (preferably new) diskette for use as a backup diskette. Leave the write-protect tab off.

5. After the TRSDOS prompt message, enter "BACKUP." The Radio Shack BACKUP program should load and display the message

TRSDOS BACKUP UTILITY VER 2.X
SOURCE DRIVE NUMBER?

6. Enter "0" for the source drive number.

7. The BACKUP program will then ask

   DESTINATION DRIVE NUMBER?

8. Enter "0" for the destination drive number if you have a one-drive system, or "1" if you have a multi-drive system. If you have a multi-drive system, insert the backup diskette in drive 1.

9. The BACKUP program will then ask

   BACKUP DATE (MM/DD/YY)?

10. Enter six digits separated by slashes for the month, day, and year.

11. The BACKUP program will now display a flashing message

    INSERT SOURCE DISK (ENTER)

12. As you already have the muMATH/muSIMP disk in drive 0, press ENTER.

13. The BACKUP program will now display the flashing message

    INSERT DESTINATION DISK (ENTER)

14. If you have a one drive system, remove the muMATH/muSIMP diskette from drive 0, and put in the backup diskette. Then press ENTER. If you have a multi-drive system the backup diskette is already in place in drive 1.

15. The BACKUP program will then complete the copying procedure. (For a single-drive system, the user must alternately load the source and destination diskettes as prompted by BACKUP.) At the end of BACKUP, the message

    BACKUP COMPLETE
    HIT ENTER TO CONTINUE

will be displayed.

16. Remove the original muMATH/muSIMP diskette and store in a safe place.

How To Load muMATH/muSIMP

Loading muMATH. Take the backup copy of muMATH/muSIMP that you have just created and load by the following procedure:

1. Turn the TRS-80 system on.

2. Place the diskette into drive 0 of the TRS-80 system.

3. Reset the system by pushing the [RESET] button.

4. The TRSDOS title message should appear at the top of the screen

   TRSDOS OPERATING SYSTEM — VER. 2.X
   DOS READY

5. If you have a 32K RAM system, type "MATH32" followed by [ENTER].
   If you have a 48K RAM system type "MATH48" followed by [ENTER].

6. After about twenty seconds, the title message for muMATH should appear at the top of the screen. (There will be disk activity in the interim.)

   TRSMATH-80 1.X
   COPYRIGHT (C) 1980 MICROSOFT
   LICENSED FROM THE SOFT WAREHOUSE
   ?

7. muMATH has now been loaded and is active.

8. If you are interested in experimenting with muMATH at this point without reading further, you may input simple expressions terminated by a semicolon (";") and [ENTER]. Try some of the examples given in the "What Is muMATH/muSIMP Section," or some of your own using "+" (addition), "−" (subtraction or negation), "/" (division), "*" (multiplication), "↑" (exponentiation), and "!" (factorialization).

9. You may also wish to run a set of demonstration problems that illustrate the capabilities of muMATH. To load the problem set, type in the

following command

RDS (DEMO, INT); ENTER

There should be disk activity, and you should see the problems with their solutions appear on the display screen. To stop the display at any time, press the space bar. To restart the display press any key. Some of the sample problems may take several minutes to execute, so if it appears that "nothing is happening," be patient!

Loading muSIMP. The procedure for using muSIMP in the programming mode is given in detail in Chapter Five.

muMATH / muSIMP Line Input and Display

muMATH User Dialogue. muMATH prompts the user with a question mark ("?") to indicate that it is ready to accept the next command. The user then types an expression followed by a semicolon (";") and ENTER.

```
? 2+5 + 8 ;
```

Note that muMATH allows "free-form" input. The terms of the expression may have spaces in between if desired, or may follow each other without spaces.

After the command line has been input, muMATH processes the command and prints the answer on the following line after the character "@" to represent "@"nswer.

```
? 2 + 5 + 8 ;
@ 15
```

This dialogue can then be repeated for new commands.

Operators. Addition, subtraction, multiplication, and division within an expression are represented by the "operators" "+," "–," "*," and "/," respectively, as, for example

```
? ( (2 – 5 + 8) * 2 ) / 2 ;
@ 5
```

Notice that parentheses were used to group the terms of the expression to avoid ambiguities about the precedence of operations.

Exponentiation is represented by the up arrow character (↑) as in

```
? 5 ↑ 2 ;
@ 25
```

Commas are used to separate "arguments" within functions. We've seen some limited use of the RDS function in reading in the DEMO/INT file. Functions are covered in more detail in the muSIMP portion of this manual.

The **colon** symbol (":") is used as an "assignment" symbol to assign a result to a symbolic name. For example, we could assign the result of the following operation to the name "ANSWER" by using a colon

```
? ANSWER : 5 * 2 ;
@ 10
? ANSWER ↑ ANSWER ;
@ 10000000000
```

Under certain conditions, it's desirable to enter a command or expression without displaying the response on the screen. A **dollar sign** is used to terminate a line in place of a semicolon in this case.

```
? ANSWER : 2 * 5 $
? ANSWER ↑ ANSWER ;
@ 10000000000
```

The **dollar sign** may be used to separate any number of separate expressions or commands.

The **single quote (')** is a special operator which returns its operand without evaluating that operand. Thus, the single quote can be used to change a "bound" variable back to "unbound" status as in

```
? B:5$
? B;
@ 5
? B : 'B$
? B;
@ B
```

The double equal sign (==) is used in expression processing (see Chapter Four) to separate the sides of an equation.

```
?EQN!: 5*X – 3*X–7 == 2+4;
@ –7 + 2*X == 6
```

The percent symbol ("%") is used to enclose comments. Any string of characters within matching percent symbols is simply ignored for any computations.

```
? ANSWER : 2 * 5 % COMPUTE THE ANSWER HERE% ;
@ 10
```

Be sure to "close" the comments by a second "%," otherwise other processing is inhibited! (muMATH will simply be looking for the second "%" before processing.)

Of course, the normal TRS-80 edit functions, such as left arrow (←) for backspace, right arrow (→) for "tab," and SHIFT left arrow for "return to beginning of line" are also in force in muMATH/muSIMP.

To Stop Processing. To stop processing the more complex and time consuming input lines, hold down the CLEAR key. muMATH/muSIMP will interrupt processing and display the message

```
* INTERRUPT * CONTINUE: ENTER;
EXECUTIVE: CLEAR; SYSTEM:BREAK?
```

If you wish to continue processing the input, press ENTER. If you wish to terminate processing of the current input, press CLEAR again. If you wish to leave muMATH/muSIMP, press the BREAK key.

Chapter Two
Arithmetic Operations Using muMATH

- Simple Arithmetic Computations
- Variables and Their Use
- Computing Factorials
- Using muMATH for Binary, Octal, Hexadecimal and Base X Arithmetic
- Irrational Arithmetic and Simplification
- Arithmetic Functions and Control Variables

Simple Arithmetic Computations

This chapter will explain how to use muMATH to perform common arithmetic operations, using expressions containing the operators previously described in "muMATH/muSIMP Line Input and Display," special arithmetic functions available for arithmetic, and several "control variables" that control arithmetic simplifications.

We have already demonstrated how muMATH performs simple arithmetic computations by processing a user-input expression terminated by a semicolon and ENTER. The expression may be as long as required.

```
? 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 ;
@ 55
```

The only common arithmetic operators that differ from hand-written symbols are the asterisk ("*") for multiplication and the up arrow (↑) for exponentiation, or "raising to a power." The asterisk must be used between terms for multiplication, unlike the hand written version. The use of up arrow for exponentiation is easy to remember if you think of a "superscript" above the term or expression to be raised to the power.

```
? 1 * 2 * 3 * 4 * 5 * 6 ;
@ 720
? 200 ↑ 2 ;
@ 40000
```

Of course, when there is a question of the "hierarchy" of operations in the expressions, parentheses must be used to group the terms just as in the handwritten version. Otherwise some incorrect results will occur. For example, if we had really wanted to perform

RESULT = (7 + 8) / 2,

but had entered

```
? 7 + 8 / 2 ;
@ 11
```

muMATH would have first performed the division of 8/2 and then performed the addition.

The correct input should have been

```
? (7 + 8)/ 2 ;
@ 15/2
```

Rational Arithmetic. The answer illustrates an interesting point. muMATH performs "rational" arithmetic, reducing the results to a fraction of integer values. Decimal fraction and irrational numbers (those that cannot be expressed as an integer fraction) are never output as results and cannot be accepted as inputs. Entering the expression

```
? 1.44 + 3.66 ;
```

results in the error message

```
***SYNTAX ERROR :
.44 + 3.66 ;
```

Range of Numbers. What is the range of numbers that may be represented in muMATH? **One of the most powerful features of muMATH is that it has an almost "infinite precision," that is, it will accept up to about 611 digits in a term and output up to about 611 digits in a result.** For most calculations this is far more than adequate.

We saw a large result in some of the introductory material, but here's another example of the capability of muMATH in this regard (with thanks to our abacus instructor):

```
? 12345679012345679012345679012345679 * 9 ;
@ 111111111111111111111111111111111111111
```

Results of hundreds of digits are easily handled by muMATH, although the processing time required increases with the number of digits.

Variables and Their Use

If you are using muMATH at all, you are probably using it to do a series of calculations. Many times the result of some previous calculation is required. How is such a result set aside?

The last result is always saved as a variable called "#ANS." If, for example, we wish to square the result of the previous example we can input

```
? # ANS ↑ 2 ;
@ 1234567901234567901234567901234567898765432098765432098765432098765432
```

The #ANS result is always the last result. If another input, no matter how innocuous, is used, #ANS will be set to the value of the new result. What can we do to save more than one intermediate result, and for longer than the next calculation?

Variable Assignment. To save a number of intermediate results, a user-named variable may be used. The line input starts with the variable name, followed by a colon (":"), followed by the equation to be evaluated. The named variable is set equal to the result.

```
? ANSWER : 5 * 2 ;
@ 10
? ANSWER ↑ ANSWER ;
@ 10000000000
```

Here the result of 10 was assigned to the symbolic name ANSWER. In the second input line ANSWER operated on itself and the new result was ANSWER raised to the ANSWER power, or 10 to the tenth power.

Names can be any sequence of alphabetic characters, digits 0 through 9, or the character "#," as long as the first character of the name is alphabetic or "#." For example, A, AA, A9, A1234, #9, RESULT, VOLUME, ANGLE1, and DISTANCE are all valid names. In addition, a name may be a string of any characters within double quotes, such as "SPEED OF LIGHT."

There are certain names that are reserved by the system to represent common mathematical constants. "#PI" is used for π, "#E" for e, the base of natural logarithms, and "#I" for the imaginary number i, $i = \sqrt{-1}$.

As many variables as are required may be used in a set of calculations. Variables retain their values throughout an entire set of calculations, just as they would if an intermediate result was jotted down on paper to be used in a later calculation.

Any variable may be redefined by a new result as required. In this case, the redefinition is analogous to crossing out the old value on paper and replacing it with the value obtained from a new calculation. An example might help to make this clear:

```
? TEMP : 5 * 5 ;
@ 25
? TEMP : TEMP * 5 ;
@ 125
? TEMP ;
@ 125
```

In the above dialogue, a variable called TEMP was set equal to the result of 5 * 5 or 25. Next, TEMP was multiplied by 5, and the result of 125 was assigned to TEMP by the second assignment statement. Finally, the variable TEMP was input alone. muMATH, recognizing TEMP as a variable previously used, printed the current value of TEMP as 125.

A number of variables may be assigned the same value by multiple assignment as in

```
? A:B:C:5;
@ 5
```

Here's another sample of a dialogue using variables:

Problem: Find the distance fallen after 2 seconds by a malfunctioning disk drive dropped from the upper floor of a computer repair center. We'll use the well known formula for motion in free fall, D=V0*T+(A*T↑2)/2. A is -32 ft/sec/sec.

```
? A : –32 $
? V0 : 0 $
? D : V0*2 + A*2↑2/2;
@ –64
```

In the above, we used three variables, A, V0, and D. Two of these were initialized to constant values. The third, D, was used to hold the final result. We used the dollar sign ("$") to indicate that the first two inputs were not to be printed out.

Computing Factorials

The factorial operator ("!") is one of the "built-in" operators in muMATH. For any integer number, N, N! is the product of

```
1 * 2 * 3 ...* (N–1) * N
```

Computing the factorial of any number is easy in muMATH. To compute 100! (1*2*3*4*...*90*100), for example we have

```
? 100! ;
@933262154439441526816992388562667004907159682643
816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
```

Problem: As another example of the use of factorials, consider the following problem. A computer system user has 101 different diskettes. Two of the diskettes are "masters" containing his most valuable programs (including muMATH/muSIMP, of course). What is the probability that he can rescue the two masters from his burning computer room if he has time to save only two diskettes?

Answer: The number of combinations of N things taken R at a time is

```
C=N!/(R!*(N-R)!)
```

Using muMATH,

```
? C : 101!/(2!*(101-2)!) ;
@ 5050
```

The probability is therefore 1/5050. (The chance of rescuing the two masters is only one out of 5050.)

Using muMATH for Binary, Octal, Hexadecimal or Base X Arithmetic

muMATH normally operates in base ten (decimal) arithmetic. However, the input and output base may be any number base from two through thirty six. The bases of two (binary), eight (octal), and sixteen (hexadecimal) are important to those interested in computer science or digital design pursuits. We'll show some examples of operations in these bases, but bear in mind that any base up to thirty six may be just as conveniently used.

The RADIX function is one of many muMATH functions that perform predefined operations based on the arguments presented in the function reference. The format of the RADIX function is

```
RADIX (N) ;
```

N is a value representing the number base that is to be used. The number base initially is decimal or base ten. To change to base 2, we would input

```
? RADIX (2) ;
@ 1010
```

The radix of the previous base is displayed. ("1010" is 10 in binary.)

All output numbers will be displayed as binary numbers; all input numbers must likewise be made up of only the binary digits 0 or 1. The following dialogue would display the decimal number 213 in base 2, base 8, and base 16:

```
? NUMBER : 213 $
? RADIX (2) $ NUMBER ;
@ 11010101
? RADIX (1000) $ NUMBER;
@ 325
? RADIX (20) $ NUMBER;
@ D5
```

The first use of RADIX changed the base from decimal to binary. NUMBER was then displayed in its binary form of 11010101.. The next use of RADIX changed the base from binary to octal. Now here's the rub . . . since the output and input base was currently in binary, the value for octal in the RADIX call had to be expressed in binary! We could not have said RADIX (8), as 8 is a non-existent numeral in binary. Instead we had to use 1000, which is a decimal 8. After 213 decimal was displayed in octal (325), the base was changed again, from octal to hexadecimal. As the current base for the third RADIX call was octal, the value of 16 had to be expressed in octal, or had to be 20, an octal 16. The value of NUMBER was then displayed in hexadecimal, D5.

It's easy to get confused over which base is currently in use. Is there a fool-proof way to get back to decimal? As 2 is a valid number in any base from 3 through 36, the following input will always let us get back to decimal from any base except base two:

```
?  RADIX   (2)       %SET BASE TO BINARY% $
?  RADIX   (1010)    %SET BASE TO DECIMAL% ;
@  2
```

The first use of RADIX changes the base to binary, while the second use changes the base from binary to 1010, which is 10 decimal expressed in binary.

Valid binary digits are 0 and 1; valid digits in base three are 0, 1, and 2; valid digits in octal are 0, 1, 2, 3, 4, 5, 6, and 7; valid digits in base 10 are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. What are valid digits for bases above ten? The valid digits for any base are digits from 0 through the digit that's one less than the base value. If the base value is greater than 10, the alphabetic symbols A through Z are used as required in lieu of numeric digits, in order to have a single character represent each digit.

In hexadecimal, then, the valid digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. In base 23 (a base used in the Alpha Centauri planetary system), the valid digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G, H, I, J, K, and L. The decimal number 213 expressed in base 23 is

```
? NUMBER : 213 $ RADIX (23) $ NUMBER ;
@ 96
```

. . . and the decimal number 1381684805 expressed in base 36 is

```
? NUMBER : 13881684805 $ RADIX (36) NUMBER ;
@ MUMATH
```

Once the RADIX is set to any particular base, muMATH can conveniently be used as a calculator in that number base. The following shows some hexadecimal calculations, for example.

```
? RADIX (16) ;
@ A
? 0FFCD + 0E345–66 ;
@ 1E2AC
```

Note that whenever alphabetic digits are input that represent digits of the number base in use, they must be preceded by a leading zero to differentiate them from variable names.

Irrational Arithmetic and Simplification

muMATH, as we mentioned earlier, uses rational arithmetic to perform all operations. Numbers must be expressed either as integers or fractions containing only integers.

In performing the rational arithmetic, fractions are always reduced to the lowest terms possible. Some examples of this are:

```
? 1/2 + 1/6          % 3/6 + 1/6 = 4/6 = 2/3 % ;
@ 2/3
? 5/125 + 1/25       % 5/125 + 5/125 = 10/125 = 2/25 % ;
@ 2/25
```

Irrational numbers cannot be entered into muMATH, but, of course, will appear after computations. Irrational numbers are also represented by integer fractions in expressions simplified as much as possible. As examples of representation of irrational results, let's look at some

calculations involving fractional powers, such as square roots. The following discussion applies to "MATH48" systems only as "MATH32" cannot process fractional powers.

```
? 4 ↑(1/2)          % NO PROBLEM HERE % ;
@ 2
? 1000000 ↑ (1/2)   % OR HERE % ;
@ 1000
? 1000 ↑ (1/2)      % =31.622776...% ;
@ 10 ↑ (3/2)
? 12 ↑ (1/2)        % =3.4641016...% ;
@ 2*3↑(1/2)
```

In the above calculations, an integer result appeared whenever possible. When the result was an irrational number, it was simplified as much as possible and the result was represented by an integer fraction.

Note that in the results above, the positive root only was represented; –2 was ignored as the square root of 4. muMATH will choose the positive real number over the negative if both exist. If only the negative real number exists, as in

```
? (–125)↑(1/3)      % CUBE ROOT % ;
@ –5
```

then muMATH will, of course, choose it.

If no real number exists, then muMATH will use the imaginary number i (–1)↑1/2 (represented in muMATH by #I) in the answer, as in

```
? (–9) ↑ (1/2)      % SQUARE ROOT of –9 % ;
@ 3 * #I
```

Fractional powers of fractions are handled with no problem in muMATH, as in

```
? (4/9) ↑ (3/2)     % 4/9 to the 1.5 power % ;
@ 8/27
```

Problem: For geometrically similar people, surface area increases as the 2/3 power of the mass. Mae wears a 1/2 square meter bikini. If Mae weighs 50,653 grams, and her look-alike mother June weighs in at 132,651 grams, what is the area of her mother's bikini?

Answer: The formula to use here is

Area of June's bikini + (132,651/50,653) ↑ (2/3)*(1/2)

using muMATH,

```
? ((132651/50653) ↑ (2/3)) * (1/2) ;
@ 2601/2738
```

Arithmetic Functions and Control Variables

There are many built-in functions available to the user. Some of these are involved in program control, while others are also usable in mathematical applications.

ABS is a function which returns the absolute value of the "argument."

```
? TERM : –56 $ ABS (TERM) ;
@ 56
```

DEN is a function which returns the denominator of the argument. When there is no denominator, a 1 is returned.

```
? DEN (56)          % 56/1 HERE % ;
@ 1
? DEN (4/3)         % DENOMINATOR HERE IS 3 % ;
@ 3
```

GCD is a function which returns the greatest common divisor of two integer arguments, the largest integer that will evenly divide both arguments.

```
? GCD (5537,2825) ;
@ 113
```

LCM is a function which returns the least common multiple of its two integer arguments, the smallest positive integer that is a multiple of both arguments.

```
? LCM (54,78) ;
@ 702
```

MIN is a function which returns the minimum of its two integer arguments.

```
? MIN (–56,–78) ;
@ –78
```

muMATH does not have a built-in MAX function.)

NUM is a function which returns the numerator of its argument. If the argument is not a fraction, the entire argument is returned.

```
? NUM (55)          % 55/1 HERE % ;
@ 55
```

Of course, any of the above functions may be used with variables just as easily as with constants:

```
? VAR1 : 2 $ VAR2 : –3 $
? NUM (VAR1/VAR2) ;
@ –2
? DEN (VAR1/VAR2) ;
@ 3
```

Two control variables are used in the arithmetic section of muMATH. Control variables are "system" variables that determine which of several processing alternatives are used in muMATH. (We will see many control variables in following chapters.)

ZEROBASE is a control variable that determines how muMATH will simplify

$$0 \uparrow \text{expression}$$

Obviously, if the expression is numeric and positive such as, $0 \uparrow 4$, or $0 \uparrow 100$, the result is 0. If a variable or expression containing variables and numeric data is used, however, muMATH will not produce a result of 0 unless ZEROBASE is set to TRUE. How do we set ZEROBASE to TRUE? Easy . . .

```
? ZEROBASE : TRUE ;
@ TRUE
```

Notice that the @nswer after changing ZEROBASE was TRUE. To examine the status of ZEROBASE at any time, simply input

```
? ZEROBASE ;
@ TRUE
```

and muMATH will type out either "TRUE" or "FALSE."

To see how ZEROBASE works, examine the following

```
? ZEROBASE : FALSE $
?0 ↑ 3 ;
@ 0
?0 ↑ A ;
@ 0 ↑A
? ZEROBASE : TRUE $
?0 ↑ 3 ;
@ 0
?0 ↑ A ;
@ 0
```

Note that when ZEROBASE was TRUE, the nonnumeric power produced a result of zero.

The second control variable is ZEROEXPT, which permits a similar simplification. If ZEROEXPT is TRUE, any expression to the 0 power is resolved as 1, as it should be for positive bases; if ZEROEXPT is FALSE, nonnumeric expressions do not resolve to 1.

```
? ZEROEXPT : FALSE $
?A ↑ 0 ;
@ A ↑ 0
? ZEROEXPT : TRUE $
?A ↑ 0 ;
@ 1
```

Chapter Three
Using muMATH to Solve Algebraic Problems

- Bound and Unbound Variables
- Evaluation of Expressions and Restoring Variables
- Four Important Control Variables in Algebraic Operations
- Other Control Variables for Algebraic Processing
- Algebraic Processing Functions

Bound and Unbound Variables

In the previous chapter, we saw how variable names could be used to represent arithmetic quantities. **When a variable is set to a specific arithmetic value, it is called a "bound" variable.** The variable INCHPM, for example, is bound to the value 39 37/100 in the following:

```
? INCHPM : 39 + 37/100;
@ 3937/100
```

What about variables that have had no previous value assigned? These variables are designated **"unbound" variables** (also known as indeterminates) and are, of course, quite common in algebraic expressions.

muMATH works with any number of unbound variables; treating them as variables that have not yet been assigned a value; in addition, muMATH will simplify expressions containing unbound variables by collecting similar terms and factors.

Suppose that we use the variable name X to represent an unbound variable.

```
? X ;
@ X
```

In @nswering the input line, muMATH simply repeated the simplest form of the input line. As X was not previously defined, the simplest form was the algebraic unknown X itself.

Now let's try a slightly more complicated expression

```
? 2*X – X↑2/X ;
@ X
```

Here muMATH processed the input line and performed collection of similar terms to reduce the input expression to a simplified form.

There are **many cases in which muMATH will automatically simplify** expressions. For example

```
? 0 + Y ;
@ Y
? Y*0 ;
@ 0 ;
? Y↑ 1 ;
@ Y ;
? 1 ↑ Y ;
@ 1
```

There are many other cases in which muMATH will not automatically perform certain transformations; these are situations in which it is not clear whether the transformations will actually be simplifications, or will result in an expression that is less convenient to work with. For example, it may be much easier to work with (X + Y)↑5 than with X↑5 + 5*X↑4*Y + 10*X↑3*Y↑2 + 10*X↑2*Y↑3 + 5*X*Y↑4 + Y↑5, especially if the value (X + Y)↑5 is used in subsequent expressions.

Because muMATH cannot automatically determine which expressions need to be expanded or transformed, it leaves the choice up to the muMATH user! To a large extent, use of muMATH in the algebraic mode involves learning how to use four control variables that define when to expand products or integer powers of sums, or other transformations. We'll see how to use these control variables shortly.

Evaluation of Expressions and Restoring Variables

Up to this point we have not really illustrated how muMATH may be used to solve typical algebraic equations. Let's see a typical example.

Problem: muMATH is being used in an interactive microcomputer installation in Carbohydrate Sam's Doughnut Shoppe. Sam wants to economize on materials and is figuring out volumes of various types of doughnuts by the formula for the volume of a torus

```
V = 2 *#PI↑2 * A * B
```

where A is the average radius of the doughnut and B is the radius of the "ring." Sam's dialogue with muMATH goes something like this:

```
? V : 2 * #PI↑2 * A * B   % GENERAL FORMULA % ;
@ 2 * #PI↑2 * A * B
? A : 2 $ B : 1/2 $       % DIMENSIONS OF GLAZED%
? V ;
@ 2 * #PI↑2 * A * B
? EVAL (V)                % EVALUATE % ;
@ 2* #PI↑2
? A : 3 $ B : 1 $         % DIMENSIONS OF CHOCOLATE %
? V ;
@ 2 * #PI↑2 * A * B
? EVAL (V)                % EVALUATE % ;
@ 6* #PI↑2
```

The first input established V as a variable representing the equation 2 * #PI↑2*A*B. Notes that the special muMATH variable #PI was used for π. Next, two values of A and B were input with a request for V on the next line. V was not evaluated with the values of A and B! It was displayed as the general formula.

The volume was finally computed with the values of A and B by the muMATH function EVAL, which forced the evaluation of V. The point here is that muMATH does not automatically reevaluate an expression because a related variable has been changed. The EVAL function forces muMATH to evaluate the expression with current values for all variables.

When two new values of A and B were input (the chocolate doughnut), V was similarly not recomputed with new values for A and B. The second EVAL call, however, did cause V to be evaluated with the two new values.

We will see in the next section that changing control variables also does not cause expressions to be automatically reevaluated. Expressions are only reevaluated on the basis of an EVAL call or when reassigned; otherwise they represent the "last evaluation."

To change a bound variable back to the unbound form, the single-quote operator (') is used. If V was computed with the last two values of A and B, for example, A and B could be changed back to their unbound forms from 3 and 1 by:

```
? A : 'A $ B : 'B $ A ; B ;
@ A
@ B
```

Four Important Control Variables in Algebraic Operations

There are a number of control variables that are used by muMATH to evaluate expressions. The four most important of these are "PWREXPD," "NUMNUM," "DENNUM," and "DENDEN." These four control variables are set in similar fashion to the arithmetic control variables ZEROBASE and ZEROEXPT — an assignment is made and the control variable is set to a numeric value as in

```
? PWREXPD : 3;
@ 3
```

The assignment is generally to 2, 3, 5, or to multiples of these primes such as 2*3 or 2*3*5 or 2*5. Prime numbers are used as they are easy to remember and can exactly specify the desired rules to apply.

The PWREXPD control variable controls expansion of integer powers of sums. Initially this variable is set to 0, so that integer powers of sums are not expanded as in

```
? PWREXPD ;
@ 0
? SAMPLE : (X+1)↑2 ;
@ (X+1)↑2
? EVAL (SAMPLE) ;
@ (X+1)↑2
```

Here, even though we used the EVAL function to evaluate (X+1) ↑2, muMATH did not expand as PWREXPD was set to 0.

Setting PWREXPD to a positive integer multiple of 2 (2, 4,...) causes "multinominal expansion" of positive integer powers (generally, numerators)

```
? PWREXPD : 2 ;
@ 2
? SAMPLE ;
@ (X+1) ↑2
? EVAL (SAMPLE) ;
@ 1+2*X+X ↑ 2
```

Here, PWREXPD was set to 2 to cause the expansion. Note that the expansion did not automatically occur after we had set PWREXPD to 2;

EVAL had to be used to cause reevaluation.

When PWREXPD is set to positive integer multiples of 3 (3, 6,...), negative integer powers of sums (generally, denominators) are expanded as for example

```
? PWREXPD : 3;
@ 3
? SAMPLE : 1/((X+1)↑2-X)↑2   % REEVALUATE ON
                              ASSIGNMENT % ;
@ 1/(-2*X*(1+X)↑2+X↑2+(1+X)↑4)
```

In the evaluation above, the denominator, being a negative integer power, was expanded . . . partially. Why wasn't (X+1)↑2 expanded at the same time? Since we called for expansion of negative integer powers, (X+1)↑2 was not expanded.

How do we expand both the positive integer powers and negative integer powers of terms in an expression? To do this we set PWREXPD equal to both an integer multiple of 2 and 3 by setting it equal to 2*3.

```
? PWREXPD : 2*3;
@ 6
? SAMPLE : (1+X)↑3 / (1+X+Y)↑2 % REEVALUATE
                                 ON ASSIGNMENT % ;
@ (1+3*X+3*X↑2+X↑3) / (1+2*X+2*Y+2*X*Y+ X↑2+Y↑2)
```

The other three control variables discussed in this section — NUMNUM, DENNUM, and DENDEN — are somewhat related. All three control either distribution of terms or factoring of terms in expressions. Here, as in PWREXPD, the control variable is set to a value by an assignment input line. The value used in the assignment determines the types of terms that will be distributed or factored.

If the control variable is set to a positive value, distribution of terms is performed; if the control value is set to a negative value, factoring is done.

Meaningful values for the three control variables are multiples of 2, 3, and 5. These values represent actions as follows:

Prime	Meaning
2	Distribute or factor numerical expressions
3	Distribute or factor other non-sums
5	Distribute or factor sums

NUMNUM controls distribution or factoring of factors in the numerator of an expression. If expression OUCH is 3*X*(1+X)*(1–X), then terms will be distributed as shown below for various values of NUMNUM:

NUMNUM	EVALUATION OF OUCH
0	3*X*(1+X)*(1–X)
2	X*(1–X)*(3+3*X)
3	3*(1–X)*(X+X ↑ 2)
5	3*X*(1+X–X*(1+X))
6	(1–X)*(3*X+3*X↑ 2)
10	X*(3+3*X+X*(–3–3*X))
15	3*(X–(X ↑ 2+X ↑3)+X ↑2)
30	3*X–3*X ↑ 3

Note that all of the above values called for distribution of terms, as NUMNUM was set to a positive value. A negative value for NUMNUM produces a factorization as in

```
? NUMNUM: –6 $
? (3*X + 3*X↑2)*(1–X);
@ 3*X*(1 + X)*(1–X)
```

However, factorization of sums into products of sums, which would correspond to NUMNUM being a negative multiple of 5, is not built into muMATH.

DENDEN controls the distribution or factoring of factors in the denominator of an expression over, or from, a sum in the denominator. Setting DENDEN to 2, for example, distributes numerical expressions

```
? DENDEN : 2 $
? (1/3) * (1/(B + C)) ;
@ 1/(3*B + 3*C)
```

Using negative values, of course, causes factorization analogous to NUMNUM.

Control variable DENNUM controls distribution or factoring of factors in the denominator of an expression over, or from, a sum in the numerator. If DENNUM is set to 6, for example,

```
? DENNUM : 6 $
? (B + C)/A ;
@ B/A + C/A
```

Setting DENNUM to a negative value causes factorization, which corresponds here to placing expressions over a common denominator.

What are the best values to be used for the four control variables above? A good question. May I have the envelope please . . . The answer is, of course, that it depends upon the algebraic problem involved. Some experimentation will be necessary to become comfortable in using the control variables for your purposes. In the meantime, we'll provide the following suggestions:

PWREXPD:0; NUMNUM:DENDEN:DENNUM:6; Good for general purpose work or to view preliminary results.

PWREXPD:6; NUMNUM:DENDEN: 30; DENNUM: -30; Fully expanded numerator over fully expanded common denominator.

PWREXPD:0; NUMNUM: 30; DENDEN: -6; DENNUM: -30; Semi-factored numerator over a semi-factored common denominator.

PWREXPD:2; NUMNUM:30; DENDEN: -6; DENNUM: -30; Good compromise between the advantages of expansion and factoring.

PWREXPD: 6; NUMNUM: DENDEN: DENNUM:30; Good for series expansions or partial fractions.

Note however, that muMATH does not automatically cancel non-numeric factors that are not explicitly present in both the numerator and the denominator.

Other Control Variables for Algebraic Processing

There are five other control variables that can be used to control processing of algebraic expressions — "ZEROEXPT," "ZEROBASE," "NUMDEN," "BASEXP," and "EXPBAS."

The first two, **ZEROEXPT and ZEROBASE,** operate exactly the same as the description under "Arithmetic Operations." ZEROEXPT controls simplification of an expression raised to the zero power; ZEROBASE controls simplification of zero raised to a power represented by an expression.

NUMDEN operates similarly to NUMNUM, DENDEN, and DENNUM. In this case the variable controls the distribution of factors in numerators over the terms of denominator sums. Here again, NUMDEN can be set to 2, 3, or 5,

or multiples of these primes to control distribution of numerical expressions, other non-sums, or sums.

Here's an example of NUMDEN when it is set to 6:

```
? NUMDEN : 6$
? A /(B + C) ;
@ 1/(B/A + C/A)
```

Denesting of denominators is accomplished by setting NUMDEN to a negative value for factoring.

Here again, experimentation with various expressions and NUMDEN values will let you see how NUMDEN can be used for your particular applications.

The BASEXP control variable controls distribution of a BASe over terms in an EXPonent which is a sum, or the reverse process, a collection of similar factors. Suppose, for example, we have A↑(B+C). Setting BASEXP to 3 causes distribution of terms as follows

```
? BASEXP : 3 $
? A↑(B + C) ;
@ A↑B * A↑C
```

The reverse process, when BASEXP is set to a negative value, collects similar factors

```
? BASEXP : –3 $
? A↑B * A↑C ;
@ A↑(B + C)
```

The EXPBAS control variable controls the distribution of EXPonents over BASes which are products. The expression (A*B)↑C, for example, is transferred by EXPBAS as follows

```
? EXPBAS : 3 $
? (A*B) ↑ C ;
@ A↑C*B↑C
```

Here again, the reverse process with negative values of EXPBAS collects bases which have similar exponents of the specified type as in

```
? EG : 2↑X * 3↑X + (1+X)↑(1/2)*(1–X)↑(1/2) – (1–X↑2)↑(1/2) $
? EXPBAS: –6 $ NUMNUM: 30 $ EVAL (EG) ;
@ 6↑X
```

Values that may be used for BASEXP and EXPBAS are multiples of the primes 2, 3, and 5; the value used, of course, controls the type of distribution or factoring used as described in the table on page 49.

Algebraic Processing Functions

There are a number of built in functions in muMATH that help in algebraic manipulations.

Three functions are related to control variable settings. EXPAND, EXPD, and FCTR all evaluate an expression with specific settings of PWREXPD, NUMDEN, NUMNUM, DENDEN, DENNUM, BASEXP, and EXPBAS.

EXPAND requests full expansion with fully distributed denominators, bases, and exponents. Control variable settings are PWREXPD:6; NUMDEN:0;NUMNUM:DENDEN: DENNUM:BASEXP:EXPBAS:30;.

```
? EXPAND (((1+X)/(1–X))↑2) ;
@ (X/(1–X)+1/(1–X)↑2
```

EXPD fully expands over a common denominator. Control variable settings are the same as EXPAND, except that DENNUM is set to –30.

```
? EXPD (1/(X+1) + (X+1)↑2) ;
@ (2+3*X+3*X ↑ 2+X ↑ 3)/(1+X)
```

The third function of this type is FCTR. It semi-factors over a common denominator using control variable settings as follows: NUMNUM: DENDEN:–6; DENNUM:BASEXP:EXPBAS:–30; PWREXPD:NUMDEN:0;

The NUM and DEN functions are identical to the NUM and DEN arithmetic functions (see "Arithmetic Functions").

The last function in this section is EVSUB. EVSUB uses a function named SUB which is used in the form

```
SUB(exp1,exp2,exp3)
```

The SUB function SUBstitutes expression 3 (exp3) for expression 2 (exp2) in expression 1 (exp1). EVSUB (exp1, exp2, exp3) is equivalent to

EVAL (SUB(exp1,exp2,exp3))

and forces the substitution and then evaluation. As an example, suppose that we have variable LAST set equal to X↑4 + X↑3–6*X↑2 – 4*X + 1. We can evaluate the expression with X=5 by the following dialogue

```
? LAST: X↑4 + X↑3 –6*X↑2 –4*X + 1 $
$ EVSUB (LAST,X,5);
@ 581
? LAST;
@ 1–4*X–6*X↑2 + X↑3 + X↑4
```

Notice that the evaluation did not reassign a new expression to LAST with X=5, but temporarily evaluated the expression. EVSUB can therefore be used to advantage to look at reevaluations without changing the basic expression.

Chapter Four
muMATH in Higher-Level Math Processing

- Use of muMATH In Higher-Level Math
- Equation Processing
- Logarithmic Simplifications
- Trigonometric Processing
- Differentiation and Integration

Use of muMATH in Higher-Level Math

We have already discussed the most difficult part of muMATH usage — the use of algebraic control variables, evaluation, and substitution. To a large extent, use of muMATH in higher-level math is straightforward, assuming that you know the relevant math. In the following sections we'll discuss the use of a few additional control variables and functions related to equations, logarithms, trigonometric functions, and calculus.

If you have "MATH32," then you will be able to process equations, logarithms, and trigonometric functions, but not perform calculus operations. The "MATH48" implementation permits all processing including calculus.

You have not had to load in additional source files from diskette in use of muMATH up to this point. From here on, however, it may be necessary to utilize the RDS function to load source file modules that implement specific sections of muMATH.

The following shows what source file needs to be loaded by RDS and when, along with the source file name and format to be used:

Processing	Memory	Source File
Equations	32/48	None required
Logarithmic simplifications	32/48	RDS(LOG,ALG) ;
Trigonometric simplifications	32/48	RDS(TRGPOS,ALG) ;
Further trig simplifications	32/48	RDS(TRGNEG,ALG) ;
Partial derivatives	48	None required
Indefinite integration	48	None required

TRGNEG should always be loaded after TRGPOS.

Each source file is an ASCII file of functions related to the processing required. The file will take some time to "compile" as it is read. If the variable ECHO is true, the text of the file will be displayed on the screen as the file loads.

Equation Processing

Equation processing is included in both the 32K and 48K versions of muMATH. It allows a complete algebraic equation to be entered into muMATH and processed by assignment, addition, subtraction, multiplication, and division of terms, squaring, and so forth. An example follows:

```
? EQN1 : 5*X–3*X–7 = = 2+4 ;
@ –7 +2*X = =6
? EQN1+(7 = = 7) ;
@ 2*X = = 13
? #ANS/2;
@ X = = 13/7
```

Note that the special double equal operator (= =) was used in the equation to separate the sides of the equation.

The two sides of the equation are automatically simplified according to the current control variable settings. muMATH will never, however, attempt to prove or disprove that the equation is an identity or has a solution. It also will not automatically shift terms from one side of the equation to the other.

Terms may be added or subtracted to both sides of the equation by using the double equal sign with the terms to be used, as shown in the example above.

Terms may be multiplied or divided by simply multiplying the variable; both sides will automatically be processed.

```
? EQN2: 5/X = = 12;
@ 5/X = = 12
? EQN2 : EQN2*X;
@ 5 = = 12*X
? EQN2/12;
@ 5/12 = = X
```

Raising both sides of the equation to an integer or fractional (MATH48 only) power may also be done in similar fashion.

Logarithmic Simplifications

Source file LOG/ALG (loaded by the command RDS(LOG,ALG)) provides logarithmic simplifications, expansion of logarithms, or collections of logarithmic terms.

There is one log function with three variations. The general log function is

LOG(expression,base)

which computes the logarithm of the given expression with the specified

base. As usual, muMATH makes no attempt to approximate irrational logarithms.

When LOG is used without a base, the current value of a control variable, LOGBAS, is used in place of the second "base" argument. LOGBAS is initially set to the base of natural logarithms, #E, but may be changed to any base value.

LOG(expression)

A third variation of the log function is

LN(expression)

This form uses #E as a base and is equivalent to LOG(expression,#E).

Automatic simplifications that occur are as follows:

```
? BASE↑LOG(EXPRESSION, BASE) % SIMPLIFICATION 1 %;
@ EXPRESSION
? LOG(1,BASE)                % SIMPLIFICATION 2 %;
@ 0
? LOG(BASE,BASE)             % SIMPLIFICATION 3 %;
@ 1
? LOG(BASE↑EXPRESSION, BASE); % SIMPLIFICATION 4 %;
@ EXPRESSION
```

Control variable LOGEXPD controls expansion of logarithms (positive values) or collection of logarithms (negative values) and base conversion. As in other control variables of this type, LOGEXPD may be set to multiples of the primes 2, 3, or 5.

If LOGEXPD is a positive multiple of 2, LOG(expression, base) is changed to LN(expression)/LN(base) when the base argument is not already #E.

If LOGEXPD is a positive multiple of 3, LOG(expression↑exponent,base) is transformed to exponent*LOG(expression,base).

If LOGEXPD is a positive multiple of 5, LOG(multiplier*multiplicand,base) is transformed to LOG(multiplier, base) + LOG(multiplicand,base) and LOG(dividend/divisor, base) is transformed to LOG(dividend,base) + LOG(1/divisor,base).

Negative multiples of the prime values cause the reverse collection process.

Examples of logarithmic operations can be seen in the demonstration file, DEMO/INT (loaded by RDS(DEMO,INT)).

Trigonometric Processing

Source files TRGPOS/ALG and TRGNEG/ALG (loaded by RDS(TRGPOS,ALG) and RDS(TRGNEG,ALG), respectively) provide trigonometric transformations. Both use a control variable called TRGEXPD, which controls expansion and factoring of trigonometric terms.

In general, TRGPOS controls expansion of trig terms, while TRGNEG controls factoring and collection of trig terms. The full capabilities of both files are preserved if TRGNEG is loaded after TRGPOS. If TRGPOS is loaded after TRGNEG, some memory space is saved at the expense of destroying the angle reduction capabilities of TRGNEG.

Both files use the common trigonometric symbols for trig functions — SIN, COS, TAN, CSC, SEC, and COT. Each of these symbols is used with an expression enclosed by parentheses. For example, to find the sine of $25\pi/4$, the input would be

```
? SIN(25* #PI/4);
@ 1/2↑(1/2)
```

As usual, muMATH makes no attempt to approximate irrational trig expressions.

The unbound "system" variable #PI is used; the user may redefine #PI (by assignment) to a rational approximation.

Angles are measured in radians.

TRGPOS simplifies SIN(0) to 0 and COS(0) to 1. Other simplifications relate to the symmetry of the trig functions; SIN(-X) is simplified to -SIN(X), and so forth.

TRGPOS Operations. TRGPOS uses control variables TRGEXPD and TRGSQ for trigonometric transformations.

TRGSQ can be set to 1, -1 or 0 and controls the following transformations:

When TRGSQ= 1: For integer n with absolute value of n greater than 1 and for all u, COS(u)↑n is transformed to COS(u)↑REMAINDER(n,2)*(1-SIN(u)↑QUOTIENT(n,2))↑2.

When TRGSQ=−1: SIN(u)↑n is transformed to SIN(u)↑REMAINDER(n,2)*(1−COS(u)↑QUOTIENT(n,2))↑2.

When TRGSQ=0: Neither of the above transformations is used on expressions.

TRGEXPD can be assigned multiples of the primes 2, 3, or 5. TRGEXPD operates in conjunction with the algebraic control variables previously discussed (PWREXPD, NUMNUM, DENDEN, DENNUM, NUMDEN, BASEXP, EXPBAS). The following actions occur for the various settings of TRGEXPD:

MULTIPLE	ACTION
2	Tangents, cotangents, secants, and cosecants are replaced by corresponding expressions involving sines and cosines.
3	Integer powers of sines and cosines are expanded in terms of sines and cosines of multiple angles.
5	Products of sines and cosines are expanded in terms of angle sums.

Settings of TRGEXPD: NUMNUM: DENDEN:30 $ PWREXPD:6 $ DENNUM: −30; will prove helpful in evaluating trig identities. A TRGEXPD setting of 30 has the effect of "linearizing" trigonometric polynomials, thus facilitating harmonic or Fourier analysis.

TRGNEG Operations. TRGNEG provides further trigonometric simplifications. Sines and cosines of angles that are numeric multiples of π are reduced to equivalent sines and cosines in the range of 0 through $\pi/4$. After this reduction, sines and cosines of 0, $\pi/6$, and $\pi/4$ are reduced to their numeric equivalents.

Other simplifications include trig functions involving their own inverse functions (SIN and ASIN (arc sine) for example) and products of trig functions (SEC(X)*COS(X) to 1, for example).

TRGEXPD can be assigned multiples of the primes 2, 3, 5, and 7 as follows:

MULTIPLE	ACTION
-2	Negative powers of tangents, cotangents, secants, and cosecants are replaced by positive powers of the corresponding reciprocal trig functions.
-3	Sines and cosines of multiple angles are expanded in terms of sines and cosines of non-multiple angles.
-5	Sines and cosines of angle sums and differences are expanded in terms of sines and cosines of nonsums and nondifferences. (Use NUMNUM=6).
+7	Sines and cosines are converted to complex exponentials.

TRGNEG contains an additional function with the same name as the control variable TRGEXPD. The format of the function is TRGEXPD (expression, integer). The function TRGEXPD provides the user with a way to evaluate the expression as if the control variable TRGEXPD had been changed, without actually changing the control variable itself. This is handy in evaluation of the expressions without running the risk of irreversible transformations.

TRGPOS and TRGNEG can be used in complementary fashion by changing the value of TRGEXPD from negative to positive to use the transformation capabilities of both files.

Examples of trigonometric operations can be seen in the demonstration file, DEMO/INT (loaded by RDS(DEMO,INT)).

Differentiation and Integration

If you have a 48K RAM system, muMATH contains the necessary modules for performing differentiation and integration; if you are using the 32K version of muMATH, calculus operations are not permitted.

If logarithms or trigonometric expressions are involved in the differentiation or integration, source files LOG/ALG, TRGPOS/ALG, or TRGNEG/ALG should also be loaded.

Differentiation. There is one function used in differentiation, DIF. DIF (expression,variable) returns the symbolic first partial derivative of

"expression" with respect to "variable." For example, to differentiate (A*X↑2) with respect to X

```
? DIF(A*X↑2,X);
@ 2*X*A
```

When the differentiation rule for a form is not known to the system, the derivative is 0 if none of the form's arguments contain the differentiation variable; otherwise, the derivative is not evaluated.

The "variable" used in the DIF function can actually be an arbitrary expression, which is then treated the same as a simple variable for differentiation purposes.

Higher order partial derivatives can be derived by **"nested"** use of DIF (at the expense of time and space) as in

```
? DIF (DIF(SIN(X*Y),X),Y);
@ –X*Y*SIN(X+Y)+COS(X*Y)
```

Indefinite Integration. The INT function is used for indefinite symbolic integration. The format of the function is INT(expression,variable). To integrate A*X + SIN(X) with respect to X, for example

```
? INT(A*X + SIN(X),X);
@ X↑2*A/2 – COS(X)
```

Note that as with most integral tables, the arbitrary constant of integration is supposed to reduce clutter.

When INT is unable to determine a closed-form integral of portions of the expression, the returned expression will contain unevaluated integrals of those portions, as in

```
? INT(X +A* # E↑X/X,X) ;
@ X↑2/2 +A*INT ( # E↑X/X,X)
```

INT uses distribution over sums, extraction of factors which do not depend upon the integration variable, known integrals of the built-in functions, a few reduction rules, and a "derivatives-divides" substitution rule. In general, it is best to use conservative control variable settings which do little to alter the form of the expression for successful integration. Integration will be successful for a modest, but useful, class of integrands.

Examples of integration and differentiation can be seen in the demonstration file, DEMO/INT (loaded by RDS(DEMO,INT)).

Chapter 5
Programming in muSIMP and TRS-80 Functions

- What Is muSIMP?
- TRS-80 System Functions
- TRS-80 Graphics Functions
- muSIMP Programming
- Advanced muSIMP Language Features
- muMATH Functions
- muSIMP Primitive Functions

What Is muSIMP?

Up to this point we have been using the muMATH system in a calculator-type mode without doing any programming. The heart of the muMATH system is a programming language called muSIMP, an acronym for microcomputer Structured IMPlementation language. muSIMP is a surface language for a modified version of LISP, a high-level language used in large symbolic math systems and artificial intelligence applications. All of the symbolic mathematics capabilities of muMATH are programmed in muSIMP. Even though the language is very powerful, it is very easy to learn to program using it.

The applicative nature of the muSIMP language requires that programs be developed in a modular and structured fashion. muSIMP consists of a few simple control structures and syntax rules and a large core of functions which operate on the muSIMP data types — integers, names, and lists. The muMATH system is incrementally built up from the basic muSIMP interpreter by adding more functions and syntax features to muSIMP. This allows new functions to be accessed as easily as the basic functions of muSIMP.

In this chapter we will show you how to add some simple functions to your muMATH system and use the graphic functions in muSIMP. The example functions that we will write illustrate some of the major features of muSIMP and add some more power to your muMATH system.

Unfortunately, the number of functions available in muSIMP and muMATH and the complexities of some of the muMATH techniques are beyond the scope of this manual. If you would like to obtain a comprehensive reference manual on muSIMP programming and operation, the muSIMP/muMATH Reference Manual may be purchased separately from Microsoft Consumer Products. NOTE: This reference manual is **very** technical and assumes prior programming knowledge.

TRS-80 System Functions

There are a few functions provided in muSIMP that help interface to the TRS-80. These functions include an alternate read select function that allows input to be read from a terminal; an exit function that returns control to the operating system; and a line printer control variable.

The RDS function (ReaD Select) allows a file to be selected as the current input instead of the console keyboard. This function is mainly used to load muMATH files and execute demonstration files. We have seen some

examples of the RDS function in Chapter 4. More formally, the function is defined as follows:

RDS (filename, ext, drive)

where **filename** and **ext** evaluate to names and **drive** either evaluates to an integer between 0 and 3 or is not present. If **drive** is omitted, then TRSDOS will search all disk drives for the file name **filename/ext.** For example,

RDS (LOG,ALG);

will read the file LOG/ALG and load all of the logarithmic and exponentional simplifications and expansions into muMATH. If a drive is specified, only that disk drive will be searched for the specified file name. For example,

RDS (DEMO,INT,1);

will read the file DEMO/INT:1 .

Normally control of the current input file is done through the use of the function RDS as described above. However, after a file has been opened and made current, control can be returned to the console keyboard without closing the input file, simply by setting the value of the variable RDS to FALSE (i.e. RDS : FALSE;). A subsequent non-FALSE assignment to RDS will then return control to the point in the opened disk file at which the reading was suspended. If RDS () with no arguments is entered, then the variable with the name RDS is set to FALSE.

If the disk file specified is not found, then a TRSDOS error message is generated and control returns to the muMATH command level with a question mark prompt. If a disk file is the current input file and the EOF (end of file) is detected, an error message is displayed, the console is made the current input file, and an error-options trap occurs.

If the value of the name ECHO is non-FALSE, the characters being read from the current disk input file are echoed on the display.

Comments may be placed in input files if they are delineated by per cent signs as follows:

% This is a comment between per cent signs %

The corresponding function WRS (WRite Select) controls the destination of the **output**. If the WRS function is called, then the output is redirected

towards the new file. For example,

```
WRS (SAMPLE,OUT,1);
```

causes all subsequent output to be made to the file SAMPLE/OUT:1.

When the variable LPRINTER is non-FALSE, all display output is also echoed to the line printer.

The SYSTEM () function terminates the muMATH session and returns to TRSDOS.

TRS-80 Graphics Functions

In order to use the TRS-80 more effectively, graphics functions have been added to the muSIMP language. The graphics functions are very similar to those available in Level 2 and Disk BASIC. The first two functions are character oriented functions — clear screen and cursor positioning. The other three functions control the screen graphics.

The CLS () function clears the display and leaves the cursor at the home position. CLS () returns the value FALSE. For example, if you entered

```
? CLS () ;
```

the screen would be erased and FALSE would be displayed in the top left corner of the screen. Normally, this function would be used in another function and not at the command level.

The CURSOR (**row,col**) function repositions the cursor to the specified position on the screen. The range for the **row** parameter is 0 to 15 and the range for the **col** parameter is 0 to 63. If either of the parameters is out of range, the cursor is not repositioned and the function returns FALSE. If the parameters are in range, then TRUE is returned.

The three graphics functions have counterparts in BASIC; however, in muSIMP they are functions. This simply means that they always return results. The three functions are POINT(**x,y**), SET(**x,y**), and RESET(**x,y**). The range of the **x** parameter is 0 to 127 and the range for the **y** parameter is 0 to 47. If any of these three functions is called with a parameter that is out of range, the value FALSE is returned. This enables you to determine in a program if your graphics are out of range.

The POINT (**x,y**) function tests the graphics **x,y** on the screen and returns a 1 if it is set and a 0 if it is not set.

The SET (**x,y**) function returns as its result the value of POINT (**x,y**) before it sets the point **x,y** on the screen.

The RESET (**x,y**) function returns as its result the value of POINT (**x,y**) before it resets the point **x,y** on the screen.

muSIMP Programming

In this section the basic concepts of programming in muSIMP are defined. The muSIMP language is very simple to learn because there are only a few syntax rules. First, the various operators used in constructing expressions are reviewed.

Operators

Expressions are written in muSIMP using the standard concepts of precedence with mathematical operators. Parentheses can be used to clarify or change the order of evaluation of an expression. The arithmetic operators allowed are addition (+), subtraction (–), multiplication (*), division (/), exponentiation (↑), and factorialization (!). The factorial operator is a unary operator that follows its operand as in normal mathematical notation. Thus, 20! is written as 20! in muSIMP.

The assignment operator (:) allows variables to be assigned the result of an expression. Since assignment is performed by an operator, the result of the assignment is the value of the assignment. This allows multiple assignments to be made as follows

A : B: 0 or A : (B : 0)

In addition to the arithmetical operators there are comparison and relational operators. The comparison operators are equal (=), less than (<), and greater than (>). The logical operators are NOT, AND, and OR. The use of these operators is the same as in BASIC and the result of these operators is either TRUE or FALSE.

The following are some examples of valid muSIMP expressions:

```
COMBINATIONS : N! / (M! * (N-M)!)

ENERGY : MASS * C↑2
```

Uses of the comparison and relational operators will be shown in conjunction with writing muSIMP functions.

Function Definition

muSIMP functions are very easy to define. The skeleton for any function consists of the FUNCTION keyword, the name of the function, the function arguments, the function body, and the function end. This function skeleton is shown below:

```
FUNCTION name (argument list),
  task1,
  task2,
  ...
  taskn,
ENDFUN
```

The function body is a series of muSIMP "statements" or tasks that perform the desired action. Each task has a value and the result of the function is the value of the last task executed. Functions are entered into the muSIMP interpreter the same way as expressions. As a note, do not forget the comma following the argument list. Thus, when the ? prompt is printed, the function can be entered. For example, the following is a function definition for the area of a circle

```
? FUNCTION AREA (RADIUS),
        # PI * RADIUS↑2,
  ENDFUN $

?
```

The dollar sign ($) after the ENDFUN terminates the input. A semi-colon (;) could also have been used, printing the name of the function as the result of defining the function.

Functions with more than one argument are defined simply by adding more arguments to the argument list. A function with two arguments for determining combinations of N objects taken M at a time can be defined as follows

```
? FUNCTION COMBINATIONS (N, M),
        N! / (M! * (N-M)!),
  ENDFUN $
```

A function with no arguments can be defined by using an empty argument list as follows:

```
FUNCTION NOARGS (),
  ...
ENDFUN $
```

When a function is called, the old values of the formal parameters in the function's argument list are saved before the function body is executed and restored after the function is executed. All values passed to a FUNCTION are passed by value; that is, all arguments are call-by-value arguments. muSIMP has another construct for using call-by-name arguments.

Local variables may be declared for a FUNCTION by adding extra parameters to the argument list. These extra arguments are initialized to FALSE.

In order to build more complicated functions, we need conditional statements and looping statements. The conditional language construction in muSIMP is the WHEN ... EXIT construct. This construct not only conditionally executes a body of code, but it also exits the current control block.

More formally, WHEN is the leading keyword of the conditional-exit control construct, which has the general form

WHEN expression1, expression2, ... EXIT

If expression1 evaluates to FALSE, then evaluation proceeds directly to the point immediately following the matching EXIT. Otherwise, the expressions between expression1 and the matching EXIT, if any, are successively evaluated, after which evaluation proceeds to the point immediately following the next delimiter ENDLOOP, ENDBLOCK, ENDFUN, or ENDSUB.

For our purposes now we will only concern ourselves with the ENDFUN delimiter.

For example, in defining a function which finds the maximum of two arguments, we need to test the values of the arguments and return the appropriate answer. The MAX function can be defined as follows:

```
FUNCTION MAX (A, B),    % Find the maximum of A and B %
  WHEN A < B, B EXIT,
  A,
ENDFUN $
```

The first line defines the function name to be MAX and the two arguments to be A and B. The second line tests A against B, and if A is less than B, the result of the function is B. If A is not less than B, then the third line evaluates to the value of A and the function is exited with A as the result.

Functions in muSIMP are also recursive which means that they can call themselves in order to compute a result. One of the simplest examples of a recursive function is the factorial function. The normal definition is

$$N! = N * (N-1)* \ldots * 2 * 1 .$$

The recursive definition is

```
N! = 1                  for N = 0, and
N! = N * (N-1)!         for N > 0.
```

The second definition leads to the following function FACT

```
FUNCTION FACT (N),
  WHEN N=0, 1 EXIT,
  N * FACT (N-1),
ENDFUN $
```

Notice the similarity between the recursive mathematical definition and the muSIMP definition. This correspondence between mathematics and muSIMP programming makes concepts easier to understand and apply.

The next control construct available in muSIMP is the BLOCK ... ENDBLOCK construct. This allows blocks to be defined within functions so that WHEN ... EXIT constructs do not always cause functions to be exited. In other programming languages this construct often appears as an IF ... THEN ... ELSE construct. The muSIMP syntax for the BLOCK control construct is as follows:

```
BLOCK
  WHEN ... EXIT,
  ...
ENDBLOCK
```

As indicated the first task within a block must be a conditional-exit. Since other tasks or expressions after the first WHEN ... EXIT can also be WHEN ... EXIT control constructs, blocks provide a generalization of other programming languages' "case" or "if-then-else" constructs. The evaluation of tasks within a block proceeds sequentially unless a conditional exit causes evaluation to proceed directly to the point after the matching delimiter ENDBLOCK. The value of a block is that of the last expression evaluated within the block.

muSIMP provides a loop control construct that evaluates its tasks sequentially and starts over when the last task is evaluated. A non-FALSE WHEN ... EXIT conditional will cause the loop to be terminated. The syntax for this loop construct is as follows:

```
LOOP
  task1,
  task2,
  ...
  taskn
ENDLOOP
```

Evaluation repetitively cycles through the sequence of tasks until a conditional exit causes control to proceed directly to the point following the matching delimiter ENDLOOP. The value of the LOOP construct is that of the last task evaluated therein. Note that if a WHEN ... EXIT construct is not used within a LOOP ... ENDLOOP, the evaluation can never terminate, resulting in an infinite loop.

The last control structure available in muSIMP is the SUBROUTINE. Subroutines are very similar to functions in syntax and use. The major difference is that the arguments are all call-by-name. This means that arguments are passed unevaluated to the subroutine body. The syntax for SUBROUTINE is the same as FUNCTION, i.e.

```
SUBROUTINE name (argument list)
  task1,
  task2,
  ...
  taskn,
ENDSUB
```

The factorial function can be rewritten without using any recursion. The nonrecursive version illustrates iteration using the LOOP construct and some of the other features of functions. The following is one possible version of FACT.

```
FUNCTION FACT (N, RESULT), % RESULT is a local variable %
  RESULT: 1,
  LOOP
    WHEN N=0, RESULT EXIT,
    RESULT: RESULT*N,
    N: N–1,
  ENDLOOP,
ENDFUN$
```

The function operates by keeping a running product RESULT until N is counted down to 0.

The most common use for LOOPs is to simulate "for-next" loops in other programming languages. The LOOP construct is more general, but a standard transformation will convert a "for-next" loop into a muSIMP LOOP. For example,

BASIC	muSIMP
FOR I = J TO K STEP L	I: J, LOOP WHEN I>K EXIT,
for-next body	for-next body
NEXT I	I: I+L, ENDLOOP

Note that the above conversion assumes that L is positive and the values of K and L do not change inside the loop. The conversion is slightly longer when these assumptions are relaxed.

Similarly, "if-then-else" constructions can easily be converted into muSIMP BLOCKs.

BASIC	muSIMP

IF condition
  THEN
  true-statements
  ELSE
  false-statements
	BLOCK
	  WHEN condition,
	  true-statements EXIT,
	  false-statements,
	ENDBLOCK

Clearly, from the above conversions, the most powerful control structures available in BASIC can be converted into muSIMP without the loss of clarity. In fact, the GOTO-less programming enforced by muSIMP makes

programs easier to understand and the natural modularity makes muSIMP a very powerful structured programming language.

The following examples reinforce some of the programming concepts described above. The first example is a recursive function to compute Fibinacci numbers. The mathematical definition of the function is similar to that for factorials; however, the recursion is more complex. The definition is as follows:

fib (n) = 1 for n = 0 or n = 1;
fib (n) = FIB(n–1)+fib(n–2) for n > 1.

The corresponding muSIMP function can be programmed as follows:

```
FUNCTION FIB (N),
  WHEN N=0 OR N=1, 1 EXIT,
  FIB(N–1) + FIB(N–2),
ENDFUN $
```

First, try this function for small values of N (i.e. less than 10). The recursion performed in this function is slow because it requires FIB(N) calls to itself to evaluate itself. Try to program this function without using any recursion.

Special graphics functions can be constructed from the primitive graphics functions defined earlier. The following example draws a box at a specified position on the screen. This example shows how the modularity of muSIMP can simplify a more complicated task. First, we define a function BOX that clears the screen and draws the line segments to construct the box.

```
FUNCTION BOX (X1,Y1,X2,Y2),
  CLS (),                    % Clear the screen %
  HLINE (X1,X2,Y1),          % Draw bottom line %
  HLINE (X1,X2,Y2),          % Draw top line %
  VLINE (X1,Y1,Y2),          % Draw left line %
  VLINE (X2,Y1,Y2),          % Draw right line %
ENDFUN $
```

Now all that remains is to define HLINE and VLINE.

```
FUNCTION HLINE (X1,X2,Y),
 LOOP
  WHEN X1 > X2 EXIT,
  SET (X1,Y),
  X1: X1+1,
 ENDLOOP,
ENDFUN $

FUNCTION VLINE (X,Y1,Y2),
 LOOP
  WHEN Y1>Y2 EXIT,
  SET (X,Y1),
  Y1: Y1+1,
 ENDLOOP
ENDFUN $
```

Note that X1 and Y1 must be less than X2 and Y2, respectively.

The last example is a muMATH function to compute a Taylor series expansion for a given function. (This example requires MATH 48 to execute.) Using Taylor series, many functions can be approximated by easy-to-evaluate polynomials. The function TAYLOR uses several local variables including NUMNUM and DENNUM to allow temporary redefinition of the control variables.

```
FUNCTION TAYLOR (EXPN, X, A, N,
          % Local vars: % J, C, ANS, NUMNUM, DENNUM),
 NUMNUM: DENNUM: 30,
 J: ANS: 0,
 C: 1,
 LOOP
  ANS: ANS + C * EV SUB(EXPN,X,A),
  WHEN J=N, ANS EXIT,
  EXPN: DIF(EXPN,X),
  J: J+1,
  C: C * (X-A)/J,
 ENDLOOP,
ENDFUN $
```

For example,

```
?TAYLOR (SIN (X), X, 0, 10);
@ X-X↑3/6+X↑5/120-X↑7/5040 + X↑9/362880
```

The EVSUB function causes EXPN to be evaluated substituting A for X. Inside the loop higher and higher order derivatives of EXPN with respect to X are computed. The truncated Taylor series is kept by the local variable ANS which is returned as the result when the desired number of terms has been computed.

Advanced muSIMP Language Features

This section is a brief overview of some of the muSIMP language features that are beyond the scope of this manual, but of interest to those who would like to know more about the structure of muMATH.

muSIMP is a powerful surface language for a LISP-like interpreter. The syntactic and semantic extensibility of muSIMP are in a large part derived from its Pratt parser. This parser allows new operators with specific left and right binding powers to be easily added. The muMATH modules written in muSIMP are the best example of this extensibility. The standard muSIMP language only recognizes arithmetic on integers. The muMATH modules incrementally redefine how the basic mathematical operators operate on their operands. The first muMATH module added to muSIMP is a rational arithmetic package, then basic algebra is added. The remainder of the modules build on top of this framework. By understanding how muMATH works, new mathematics modules can easily be added.

The recursive nature of muSIMP is complemented by its dynamic scoping rules for variables. Shallow binding of variables and a closed pointer universe greatly add to the efficiency of the muSIMP interpreter. Functions and subroutines can also be defined as spread or no-spread. This allows functions to be passed an arbitrary number of arguments which can be selected by using various muSIMP primitive functions.

The muSIMP primitive functions consist of many LISP-like functions, including a full complement of selector, constructor, comparator, and recognizer functions. Internally, the compactifying garbage collector performs automatic, dynamic memory management on all data spaces allowing the computer to respond to queries of arbitrary difficulty. All of these features make muSIMP a serious language for programming language design, artificial intelligence applications, algorithm design, and symbolic mathematics. This power is shown in all aspects of the muMATH package.

For further detailed information the muSIMP/muMATH Reference Manual is available from Microsoft Consumer Products. This manual also details some additional mathematical capabilities not available with this package including matrix algebra, more integration power, and a trace package.

muMATH Functions

In this section brief descriptions are given for some of the most useful muMATH functions.

Basic Arithmetic Functions

ABS (expr)

ABS is a function which returns the absolute value of its argument when the argument is a rational number. Otherwise, the unevaluated absolute value form is returned.

COEFF (expr)

COEFF is a selector function that returns the coefficient (i.e. the numeric factors) of an expression which is a product; the **expr** if NUMBER (expr); otherwise, it returns 1.

DEN (expr)

DEN is a selector function which returns the denominator of its argument, returning 1 when there is none.

EVSUB (expr, subexpr, replacement)

EVSUB is a function which returns the result of evaluating a copy of its first argument, wherein each syntactic occurrence of its second argument is replaced by the third argument.

GCD (intgr1, intgr2)

GCD is a function which returns the positive greatest common divisor of its integer arguments.

LCM (intgr1, intgr2)

LCM is a function which returns the positive least common multiple of its integer arguments.

MIN (intgr1, intgr2)

MIN is a function which returns the minimum of its two integer arguments.

NUM (expr)

NUM is a selector function which returns the numerator of its argument, returning the entire argument when there is no denominator.

NUMBER (expr)

NUMBER is a recognizer function which returns TRUE if and only if its argument is an integer or rational number.

POWER (expr)

POWER is a recognizer function which returns TRUE if and only if its argument is of the form expr1 ↑ expr2.

PRODUCT (expr)

PRODUCT is a recognizer function which returns TRUE if and only if its argument is of the form expr1 * expr2. It is important to realize that quotients are represented as products involving negative powers.

SUM (expr)

SUM is a recognizer function which returns TRUE if and only if its argument is of the form expr1 + expr2. It is important to realize that differences are represented as sums involving terms having negative coefficients.

Basic Algebra Functions

EVAL (expr)

EVAL returns the evaluated and simplified expression resulting from **expr** operated on under the current control variable environment.

EXPAND (expr)

EXPAND evaluates **expr** to yield a fully expanded denominator distributed over the terms of a fully expanded numerator. The following **temporary** assignments are made:
PWREXPD: 6; NUMDEN: 0;
NUMNUM: DENDEN: DENNUM: BASEXP: EXPBAS: 30;

EXPD (expr)

EXPD evaluates **expr** to yield a fully expanded numerator over a fully expanded denominator. The following **temporary** assignments are made:
PWREXPD: 6; NUMDEN: 0; DENNUM: -30;
NUMNUM: DENDEN: BASEXP: EXPBAS: 30;

FCTR (expr)

FCTR evaluates **expr** to yield a semi-factored numerator over a semi-factored denominator. The following **temporary** assignments are made:
PWREXPD: NUMDEN: 0; NUMNUM: DENDEN: -6; DENNUM: BASEXP: EXPBAS: -30;

Logarithm Functions

LOG (expr)	LOG is used as an abbreviation for LOG (expr,LOGBAS) on input and output, where LOGBAS is a control variable initially set to #E.
LOG (expr, base)	LOG is used to represent logarithms in a given base.
LN (expr)	LN is used as an abbreviation for LOG (expr, #E).

Trigonometric Functions

The following trigonometric functions are allowed in algebraic transformations.

SIN (expr) COS (expr) TAN (expr)
CSC (expr) SEC (expr) COT (expr)

TRGEXPD (expr, intgr) TRGEXPD evaluates **expr** with the **temporary** assignment: TRGEXPD: **intgr.**

Basic Calculus Functions (not available in 32K muMATH systems)

DIF (expr, var)	DIF computes the partial derivative of **expr** with respect to **var.**
INT (expr, var)	INT computes the indefinite integral of **expr** with respect to **var.**

muSIMP Primitive Functions

The three data types available to muSIMP are integers, names, and lists (binary trees). The following primitive muSIMP functions operate on these data types.

Binary trees are the primary data structure in muSIMP. Internally, they are implemented as a network of cell pairs called nodes. Each node consists of a FIRST cell and a REST cell. The node cells can only point to other nodes, integers, or names.

Selector Functions

FIRST (expr)	FIRST returns the contents of the FIRST cell of expr.
REST (expr)	REST returns the contents of the REST cell of expr.
SECOND (expr)	SECOND returns the value FIRST(REST(expr)).
THIRD (expr)	THIRD returns the value FIRST(REST(REST(expr))).

Constructor Functions

ADJOIN (expr1,expr2)	ADJOIN creates a new cell whose FIRST cell is expr1 and whose REST cell is expr2.
LIST (expr1, ..., exprn)	LIST creates a linked list of its arguments.
REVERSE (list)	REVERSE returns the reverse of the given list.

Modifier Functions

CONCATEN (list1, list2)	CONCATEN concatenates the two lists into a single list.

Recognizer Functions

NAME (expr)	NAME returns TRUE if and only if expr is a name.
INTEGER (expr)	INTEGER returns TRUE if and only if expr is an integer.
ATOM (expr)	ATOM returns the value NAME(expr) OR INTEGER(expr).
EMPTY (expr)	EMPTY returns TRUE if and only if expr is the empty list.
POSITIVE (expr)	POSITIVE returns TRUE if and only if expr is a positive integer

NEGATIVE (expr)	NEGATIVE returns TRUE if and only if expr is a negative integer.
ZERO (expr)	ZERO returns TRUE if and only if expr is 0.

Numerical Functions

The following primitive numerical functions are defined in muSIMP for ease of explanation.

```
FUNCTION MINUS (X),
    WHEN INTEGER (X),
        –X EXIT,
ENDFUN;

FUNCTION PLUS (X, Y),
    WHEN INTEGER (X) AND INTEGER (Y),
        X + Y EXIT,
ENDFUN;

FUNCTION DIFFERENCE (X, Y),
    WHEN INTEGER (X) AND INTEGER (Y),
        X – Y EXIT,
ENDFUN;

FUNCTION TIMES (X, Y),
    WHEN INTEGER (X) AND INTEGER (Y),
        X * Y EXIT,
ENDFUN;

FUNCTION QUOTIENT (X, Y),
    WHEN INTEGER (X) AND INTEGER (Y),
        WHEN Y = 0, zero-divide error trap EXIT,
      WHEN POSITIVE (Y), floor (X/Y) EXIT,
        ceiling (X/Y) EXIT,
ENDFUN;

FUNCTION MOD (X, Y),
    X – (Y * QUOTIENT(X,Y)),
ENDFUN;

FUNCTION DIVIDE (X, Y),
    WHEN INTEGER (X) and INTEGER (Y),
        ADJOIN (QUOTIENT(X,Y), MOD(X,Y)) EXIT,
ENDFUN;
```

Printer Functions

PRINT (expr)

PRINT prints names and integers according to the current radix to the current output file. Lists are printed in the standard LISP form.

NEWLINE ()

NEWLINE prints a carriage return and line feed to the current output file.

SPACES (expr)

SPACES prints **expr** spaces.

PRTMATH (expr, rbp,lbp)

PRTMATH prints **expr** in standard mathematical form and surrounds it within parentheses if the leading operator in **expr** has a left binding power less than or equal to **rbp,** or a right binding power less than **lbp.** Normally called with **rbp** and **lbp** equal to 0.

The binding powers of the muSIMP / muMATH operators are shown below:

Category	Operator	LBP	RBP
Ordering	(	200	0
Assignment	:	180	20
Numerical	!	160	0
	↑	140	139
	*	120	120
	/	120	120
	+	100	100
	-	100	100
Comparison	=	80	80
	<	80	80
	>	80	80
Logical	NOT	70	70
	AND	60	60
	OR	50	50

Note: When "+" and "-" are used as prefix operators a right binding power of 130 is used instead of 100.

Index

400 108th Ave. N.E., Suite 200
Bellevue, WA 98004
(206) 454-1315

Catalog No. 1208

Part No. 10F08

Printed in U.S.A.

MICROSOFT

muMATH 2
FOR TRS-80

MATH48

Catalog No. 1208 Part No. 13H08A

MICROSO

CONSUMER PRODUCTS

400 108th Ave. N.E., Suite 20
Bellevue, WA 98004

MICROSOFT muMATH

Problem: Find the binomial expansion of $(A + B)^5$
You Enter: PWREXPD:2; (A + B) ↑ 5;
Computer Replies: @ 5*A*B ↑ 4 + 10*A ↑ 2*B ↑ 3 + 10*A ↑ 3*B ↑ 2 + 5*A ↑ 4*B + A ↑ 5 + B ↑ 5

Problem: Integrate the expression (2X—1/X) with respect to X
You Enter: INT(2*X—1/X,X);
Computer Replies: @ X ↑ 2 — LN(X)

Problem: Add the fractions 1/3, 5/6, 2/5, 3/7
You Enter: 1/3 + 5/6 + 2/5 + 3/7;
Computer Replies: @ 419/210

You'll turn your computer into a mathematical genius with the muMATH Symbolic Math package. Arithmetic, algebra, trigonometry and calculus problems, like the ones shown here, can be solved calculator style—in seconds—with 611-digit precision. muSIMP, the language in which muMATH is written, is included, too. A superset of LISP, muSIMP is especially suited to programming interactive symbolic mathematics and other artificial intelligence applications.

The Creators

muMATH and muSIMP were developed by Dr. David Stoutemeyer and Al Rich of The Soft Warehouse, Honolulu, HI.

Microsoft Limited Warranty

Microsoft will exchange this product within one year of original purchase if defective in manufacture, labeling or packaging. Except for such replacement, the sale or use of this program is without warranty or liability. No other warranty is expressed or implied.

Microsoft Copyright

MICROSOFT
CONSUMER PRODUCTS
10800 Northeast Eighth, Suite 507
Bellevue, WA 98004
(206) 454-1315

Part No. 10G08
Made in U.S.A.